TN
1960s

MARCH 1967/MTP-58

the TREET programming system IBM 7030 implementation

E. C. HAINES

Information Sciences Department

The MITRE Corporation

INFORMATION SYSTEM LANGUAGE STUDIES NUMBER THIRTEEN...

INFORMATION SYSTEM LANGUAGE STUDIES NUMBER THIRTEEN

MTP-58

THE TREET PROGRAMMING SYSTEM: IBM 7030 IMPLEMENTATION

by

E. C. Haines

March 1967

This document has been released
for public dissemination.

MITRE Project 1108

Preface

The TREET Programming System described in this report has been used in a variety of applications: (1) A procedure for transformational syntactic analysis has been developed that will process sentences in English and provide a characterization of their underlying structure in terms of a transformational grammar stored in the system.[1] (2) An on-line programming system has been produced for working with complex symbolic data in a display environment.[2] This system has been used extensively in the development of procedures for constructing and testing grammars[3] and for the design and implementation of a system for text processing.[4] (3) Programs have been written for constructing and executing procedures and algorithms within AESOP, a display-oriented on-line information management and control system.[5]

An improved version of the TREET System is being written for IBM System/360 computers. It is operational in an off-line mode and is being used currently for programming augmented versions both of the text processing system and of the procedures for linguistic analysis.

---

[1] Walker, D. E. (Ed.) English preprocessor manual. SR-132, MITRE Corporation, 1965.
Zwicky, A. M., Friedman, J., Hall, B. C., and Walker, D. E. The MITRE syntactic analysis procedure for transformational grammars. AFIPS Conference Proceedings: Fall Joint Computer Conference, 1965, 27, 317-326.
Walker, D. E., Chapin, P. G., Geis, M. L., Gross, L. N. Recent developments in the MITRE syntactic analysis procedure. MTP-11, MITRE Corporation, 1966.

[2] Gross, L. N. On-line programming system: user's manual. MTP-59, MITRE Corporation, 1967.

[3] Chapin, P. G. Natural language processing I - transformational grammar; Natural language processing II - on-line grammar development. Films, MITRE Corporation, 1966.

[4]Walker, D. E. On-line text processing: introduction and overview. MTP-57, MITRE Corporation, 1967.

Chapin, P. G., Gross, L. N., Norton, L. M., Beller, R. J., and Browne, C. T. SAFARI, an on-line text processing system: user's manual. MTP-60, MITRE Corporation, 1967.

Chapin, P. G. SAFARI: a progress report on the development of a text processing system. Film, MITRE Corporation, 1966.

[5]Bennett, E., Haines, E. C., and Summers, J. K. AESOP: a prototype for on-line user control of organizational data storage, retrieval, and processing. AFIPS Conference Proceedings: Fall Joint Computer Conference, 1965, 27, 435-455.

Summers, J. K., and Bennett E. AESOP-A final report: a prototype on-line interactive information control system. In D. E. Walker (Ed.) Information System Science and Technology. Washington, D. C.: Thompson Book Co., 1967. Pp. 69-86.

Acknowledgments

The initial work on TREET was done under Project 7020 -- Language Processing Techniques. The incorporation and use of TREET within Project 5100-AESOP has influenced certain aspects of the implementation and provided additional resources for program development. Specific acknowledgements are made to David Ewer for programming assistance and to Renée Beller for editorial assistance in the preparation of this report.

Digitized by the Internet Archive
in 2019 with funding from
Kahle/Austin Foundation

https://archive.org/details/treetprogramming58echa

TABLE OF CONTENTS

		Page
I.	Introduction	1
II.	Data Structure	4
	A. Introduction	4
	B. Atoms	4
	1. Symbols	4
	2. Numbers	6
	3. Cards	8
	C. Lists	9
	1. Machine Storage	10
	2. Tree Representation	11
III.	Expressions	13
	A. Definition	13
	B. Rules for Grouping	14
IV.	The Defining Function -- DEF	16
	A. Definition	16
	B. Discussion	17
	1. TREET Special Characters, Special Symbols, and Spacing	17
	2. Compilation	19
	3. TREET Function Types	19
	4. Argument List and Variable List	20
	5. Label	21
	6. Statement	21
	7. Comments	21
	8. An Example Program	22

TABLE OF CONTENTS (Cont.)

		Page
V.	Predefined Functions	23
	A. List Processing Functions	23
	B. Programming Functions	23
	C. Arithmetic Functions	24
	D. Control Functions	24
	E. On-line Functions	25
VI.	TREET System Modes of Operation	26
	A. TREET Mode	26
	B. READQ Mode	26
	C. System Operation and Mode Switching	27
Appendixes		
	A. Augmented Backus Normal Form	29
	B. TREET Function Descriptions	31
	C. The Compiler	51
	D. TAP - The TREET Assembly Program	59
	E. TREET Floating Point Numbers	69
	F. Garbage Collector and List Storage	75

I. Introduction

TREET is a general purpose list-processing programming system that has been implemented for on-line and off-line use on the IBM 7030 computer. The word TREET is used to mean the following things: (1) a specific language (the TREET language); (2) an evolving abstract entity consisting of a group of languages, including the TREET language, and a set of definitions and conventions together with a philosophy relating to data structure, program structure, etc., and (3) a collection of programs for the IBM 7030.

The TREET language[1] is a list-processing language that has the capabilities of LISP 1.5, but looks more like ALGOL or FORTRAN to the user. TREET operates on list structures, and all data must be coded in that form. Within TREET, a list structure is an atom or a list of list structures. An atom is a symbol (of from 1 to 10 alphanumeric characters), a number (integer or floating point), or a card (an internal image of an 80-character IBM card). A list may have any number of members, and each member may be arbitrarily complex; a list is written by enclosing its members within parentheses.

All programs in the TREET language are written as functions. A function normally has a single value (which may be an arbitrarily complex list-structure), a unique name, and operates with zero or more arguments. A function may or may not have an effect on the system other than its value; some functions are used only for their effect, others only for their value, others for both effect and value.

---

[1] Haines, E. C. The TREET list processing language, SR-133, The MITRE Corporation, 1965.

The other languages that with the TREET language proper can be used to code functions accepted by the TREET system in the current implementation include Cambridge Polish, TAP, and SMAC. Cambridge Polish is a parenthesized prefix language, used in LISP, that is machine independent, but particularly easy for the computer to process. TAP, the TREET Assembly Program, is a basic macro-language whose syntax is machine independent. SMAC is the STRETCH (IBM 7030) macro-language, a machine-dependent language that serves as the lowest level programming language. Functions may be defined on-line in TREET, Cambridge Polish, and TAP.

The philosophy underlying TREET can be characterized in several statements. The TREET language was designed to be convenient to use, easy to learn, highly intelligible to anyone familiar with computer programming, efficient in its ability to express algorithms, efficient in operation, and easily implemented in a computer. The implementation has stressed machine independence with maximum communication among the languages used in the system and maximum access by the programmer to each of those languages.

The collection of programs that forms the TREET system for the IBM 7030 consists of a supervisor, a translator to relate the languages to each other, and a set of subroutines that provide a basic programming capability. The supervisor controls the overall operation of the system. The system translator includes a translator that converts TREET to Cambridge Polish, an interpreter that "executes" the Cambridge Polish, a compiler that converts Cambridge Polish to TAP (See Appendix C), and a macro assembler that converts TAP to machine language (See Appendix D).

The basic system subroutines include functions that can be used on-line or off-line for list processing, for performing arithmetic operations, for manipulating data in card format, for directing input/output operations, for performing standard programming operations like conditional, branch, and return, and for doing program diagnostics and debugging.

II. Data Structure

A. Introduction

The TREET system operates on list structures; all data must be in list structure form (basically similar to LISP). A list structure is an atom or a list of list structures.

<list-structure>::=<atom>|(<list-structure>*)[1]

An atom may be a symbol (1-10 alphanumeric characters), a number (integer or floating point), a card (an internal image of an IBM card), or NIL.

<atom>::=<symbol>|<number>|<card>|NIL

A list consists of a number of list cells (machine words) chained together. It is referenced by the address of the first cell in the list. Each cell then points to (i.e., addresses) the next cell in the list. The last cell in the list points to 0. The empty list is represented by address 0 which prints out as NIL and may be read in as either NIL or ( ). NIL is considered both an atom and a list. It is represented internally by register 0 which always contains a full word of zeros.

B. Atoms

1. Symbols

A symbol in TREET consists of one to ten alphanumeric characters, of which at least one is alphabetic. For every unique symbol used there is a three word entry in the symbol table, ATOMS. A symbol not already present in the system will be created when it is

---

[1] Augmented Backus notation is used throughout this document. A description of this notation may be found in Appendix A.

read in. There is space for 2000 different symbols. A symbol must be terminated at each end by a special character. The special characters are: blank, left parenthesis, right parenthesis, and comma. Extra blanks are ignored.

A symbol can be used as a datum (i.e., for itself), as a variable (in which case it has a value), or as the name of a function. The same symbol can be used in all three ways. The different uses are distinguished by context. A symbol may also have a property list. There are special functions for setting and retrieving data from property lists. A symbol freshly read in as a datum will have no value, no function definition, and no property list. A symbol is represented in memory as shown below. It is referred to by the address of the first word.

First Word (IBM 7030 index register format)

VALUE FIELD	bits 0-17	pointer to the value of the symbol, if any; bit 17 = 1 signifies that no value has been assigned.
FLAG FIELD	bits 25-27	010_2 bit 25=Ø identifies an atom
COUNT FIELD	bits 28-45	0
REFILL FIELD	bits 46-63	pointer to the value pushdown list

Second Word (mixed format)

VALUE FIELD	bits 0-17	address of function, if any
	bit 25	0 for SUBR, 1 for EXPR
	bit 26	1 denotes function defined
	bit 27	1 for type F or FU

	bit 28	1 for type U or FU in SUBR
	bits 29-45	normally not used
REFILL FIELD	bits 46-63	pointer to the property list

Third Word (A6 characters)

Contains the BCD representation of the symbol, left-adjusted in the word. Unused positions contain 00_8.

2. Numbers

There are two types of numbers in TREET: integers and floating point numbers. Normally the user need not distinguish between these, as both are accepted by all arithmetic routines.

Numbers always evaluate to themselves; thus they may not be used as variables. Floating point numbers will be rounded and printed out to the number of places specified by the variable PLACES. Rounding is done only at and for printout. On input, numbers without decimal points will be read in as integers; numbers with decimal points will be read in as floating point numbers. Arithmetic routines will convert their arguments to floating point numbers, perform their function, and, if the result is integer, will convert it to integer format.

Integers are limited to 23 bits (8,388,607) and may be positive or negative. For floating point the standard 7030 single precision floating point format is used. This provides over 14 places of precision with a range of $10^{\pm 308}$.

Input of numbers is limited to 9 digits. Output (printout) of numbers is limited to a maximum of $2^{48}-1$; no more than 14 decimal places can be printed.

a. Integers

An integer is represented by a single 7030 word and is considered to be atomic.

VALUE FIELD	bits 0-23	magnitude
	bit 24	sign
FLAG FIELD	bits 25-27	010_2 bit 25 = 0 identifies an atom
COUNT FIELD	bits 28-45	5
REFILL FIELD	bits 46-63	0

b. Floating Point Numbers

A floating point number is represented by two 7030 words and is considered to be atomic. Its address is the address of the first word. The words are in general not contiguous.

First Word

VALUE FIELD	bits 0-17	pointer to the second word
FLAG FIELD	bits 25-27	010_2 bit 25 = 0 identifies an atom
COUNT FIELD	bits 28-45	4
REFILL FIELD	bits 46-63	0

Second Word

The second word is a normal floating point word except that bit zero (exponent flag) is reserved for the garbage collector. (See Appendix F).

A more complete description of floating point numbers may be found in Appendix E.

3. Cards

A card can be thought of as similar to its physical counterpart; it can store 80 A6 characters. Cards are allocated internally in much the same way as cells. Originally all cards are placed on the free card chain. When a card is needed by a routine the first one available is removed from the chain. The reclaimer marks and collects unused cards (see Appendix F). A reclaim cycle can be called explicitly or by running out of cells or cards. A second number on the reclaim printout indicates how many cards were collected.

A card is considered to be an atom. Like all other data units in TREET it carries its own description. Most routines operating with cards actually expect a list of cards.

A card occupies eight consecutive words in core. The first value field is presently unused. The latter 7-1/2 words may be used in any way desired. Unless otherwise specified they will usually be considered to be 80 A6 characters.

C. Lists

A list is represented internally by a series of single registers - called list cells - chained together. There is one list cell for each member of a list.

A list cell is a single 7030 word in index register format. This breakdown provides for four fields: value, flag, count, and refill. The value and refill fields are used to define the list.

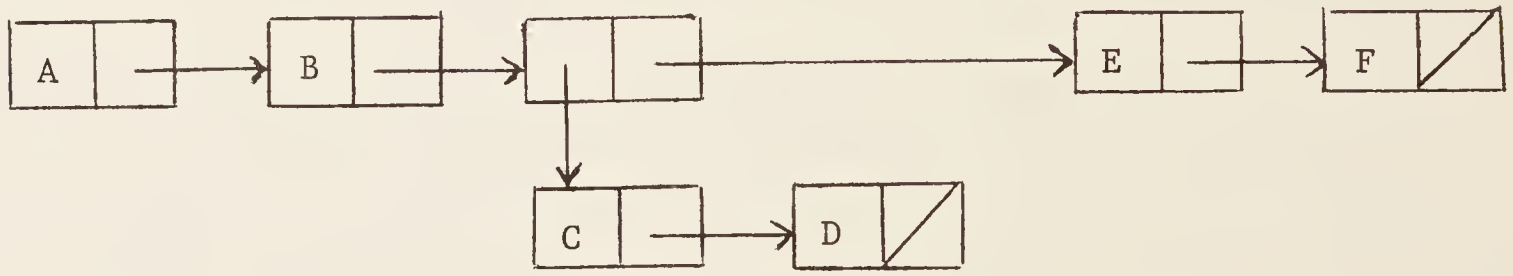
VALUE FIELD	bits 0-17	pointer to an arbitrary list structure; i.e., to an atom or another list.
	bits 18-24	0
FLAG FIELD	bits 25-27	100_2 bit 26 = 0 identifies a lost cell.
COUNT FIELD	bits 28-45	The count field may be used as a pointer or to contain some other information but it is normally left blank (zero).
REFILL FIELD	bits 26-63	pointer to next cell in list (zero in the last cell of a list).

Note that each pointer is an eighteen-bit field used to contain the address of the structure pointed to.

The value field of a list cell points to (contains the address of) a list-structure, that is, an atom, or another list. The refill field points to the next list cell in the list. The refill field of the last cell of the list contains zero; that is, it points to register Ø, which is always full zero.

A list is represented by a block diagram in which each rectangle corresponds to a list cell. Each rectangle is divided in half; the left half represents the value field, the right half, the refill field. If the value field points to an atom, then that atom is written in the left half of the box. If the value field points to a list, then an arrow is drawn from the left half of the box to the appropriate list cell rectangle. A pointer to zero is represented by a slash.

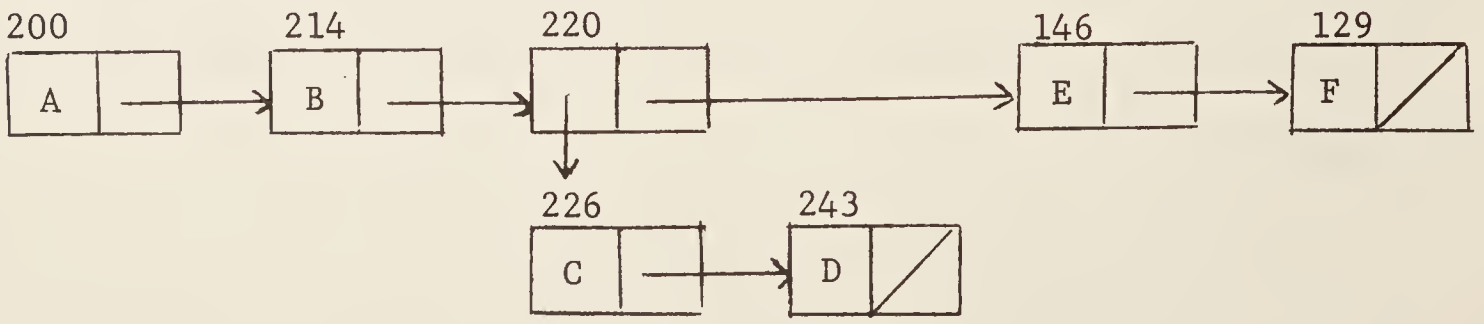
The list (A B (C D) E F) is represented in a block diagram as shown below:



The cells in a list are seldom (if ever) stored in consecutive registers in core; for this reason, the pointer from one cell to the next is necessary. Note that there is no linkage to the previous list cell.

1. Machine Storage

The actual contents of the 7 list cells for the list diagrammed above will be shown. The diagram is redrawn below with arbitrarily chosen core locations for each of the list cells written above the left half of the box.



Assume that the ATOMS table is located at 1000, and that the following correspondence between symbols and their locations pertains:

Address	Symbol
1000	A
1003	B
1006	C
1009	D
1012	E
1015	F

Then the list cells would have the following contents:

Cell Address	Value Field	Refill Field
200	1000	214
214	1003	220
220	226	146
146	1012	129
129	1015	0
226	1006	243
243	1009	0

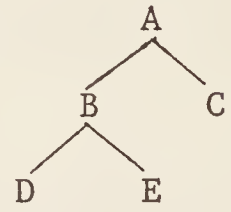
The list is referenced by the address of its first cell, in this example, by 200.

2. Tree Representation

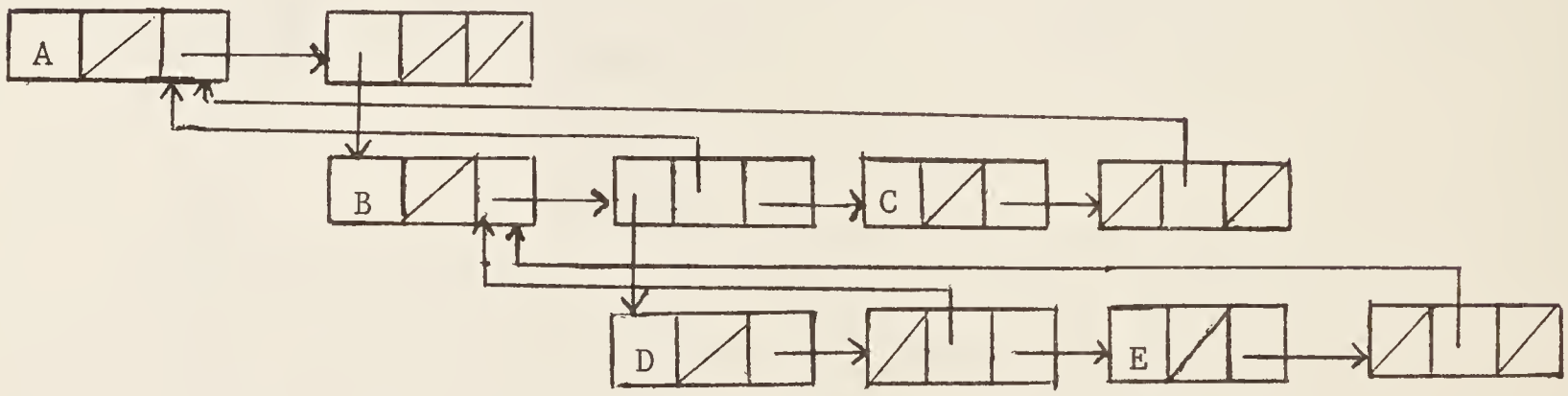
The simplified picture of a chain of list cells shows each cell as being divided into two parts: that is, the left side representing the value field, and the right, the refill field. Each list cell also contains a count field which may be used as a pointer, or may contain other information, e.g., marking certain structures. A more complete picture of a list cell would

then contain all three of these fields and could be used to show more complicated structures.

In the standard tree notation, which uses two cells per node, the first count field is available for special purposes while the second contains a pointer to the parent. For example, the tree



would be represented as:



In this example, for each pair of cells the first cell contains the value in its value field; the count field is empty, and the refill field has a pointer to the second cell. The second cell in its three fields contains pointers to the first daughter, parent, and right sister[2], respectively; when any of these relations is not present, the corresponding field is empty.

---

[2] A is the parent of nodes B and C. B is the first daughter of A, the parent of D and E and the left sister of node C.

III. Expressions

A. Definition

An expression can be defined as that which can be evaluated to produce a value. This value may be either simple or complex. The value produced in the case of a number or a literal is the expression itself. A variable, on the other hand, evaluates to another value which may be set and changed as desired. Expressions may be simple, that is, a number, a variable, a literal, or a quoted expression; they can also be functional expressions or a list made up of any type of expression including a list of expressions, and thus they can be indefinite in length and complexity. Expressions are defined recursively, as shown in the following Backus notation.

<expression>::=<simple expression>|<functional expression>|(<expression>)

<simple expression>::=<number>|<variable>|<literal>|<quoted expression>

<functional expression>::=<expression><function name><expression>|

<function name>([<expression>,]*<expression>°)|

<if expression>|<do statement>

<quoted expression>::=Q(<list structure>)

<literal>::='<symbol>'

Exceptions:

1. <expression> if NIL may be deleted in many cases.

2. <function name> if a regular function call or if Q must be terminated by the left parenthesis of its argument list. <function name> of an infix call should not be terminated by a left parenthesis.

3. IF, IFNOT, IFNULL, and DO are excluded from <function name> as used in <functional expression>.

Functional expressions can be in regular or in infix notation. Regular expressions are of the form +(A,B). The name of the function must be terminated by the left parenthesis which starts the list of operands. There can be any number of arguments within the parentheses, e.g., PRINT(ALPHA, DELTA, GAMMA). Infix notation normally has two operands which surround the function name. The name must be separated from the operands by blanks, e.g., A + B.

Functional expressions can also be in the form of an <if expression> or a <do statement>. The Backus notation of these forms follows:

<if expression>::=[IF|IFNOT|IFNULL]<expression>THEN<expression>
[ELSE<expression>]°

<do statement>::=DO(<statement>*)

<statement>::=<functional expression>;

Exceptions:

1. the first expression of an <if expression> must not contain an unenclosed THEN; in particular, if an <if expression> is desired, enclose it within parentheses. By the same token, the second expression must not contain an unenclosed ELSE. There are no restrictions on the optional third expression.

2. the final ';' generated in a <do statement> is unnecessary.

B. Rules for Grouping

The translator decodes expressions from left to right. The only type of expression which contains a precedence of functions is the <if expression>. For multiple infix expressions, grouping is as if parentheses were put in from the right to the left. For example:

A = B + 7 → (A = (B + 7))

When an <if expression> is encountered, a scan is made for the first THEN at the same level. Everything between the IF (or IFNOT or IFNULL) and the first THEN is considered to be the first expression. Similarly, everything between the THEN and the first ELSE at that level, or the end of the expression, is considered to be the second expression. If there was an ELSE, everything following it is the final expression. For example:

A = IF B LT C THEN G(D) - H(E) → (A = (IF (B LT C) THEN (G(D) - (H(E))))

IV. The Defining Function--DEF

A. Definition

All operations in TREET are functions. They may be defined by means of the function DEF. Most functions obtain all of their operating parameters from the arguments presented within the DEF function. An argument is the actual value which is given to the function at execution. This value will normally be the result of evaluating a TREET expression (F and FU functions are exceptions to this rule as are commands in READQ mode).

The arguments of DEF, taken together, define a function in the TREET language. DEF translates this definition into Cambridge Polish notation and stores it as a list structure. DEF also punches out cards on which the function is defined in compressed form. These cards are subsequently used as arguments of the BINARY function.

Functions usually map list-structure into list-structure (e.g., given a list, find its first member) but some are used for their effect (e.g., print this item on the off-line printer). Each function produces a single value which is an expression. In many cases this value is not used and always set to NIL (e.g., as in the case of PRINT). For predicate functions the response NIL means false; any other response is taken as true.

In Backus notation the DEF function is represented as follows:

<function definition>::=DEF(<name>C$^{\circ}$<type>$^{\circ}$<argument list><variable list>

[[<label>:]*<statement>]*)**

<name>::=<symbol>

<type>::=R|U|F|FU

<argument list>::=(<variable>*)

<variable list>::=(<variable>*)

<variable>::=<symbol>

<statement>::=<functional expression>;

<label>::=<symbol>

B. Discussion

1. TREET Special Characters, Special Symbols, and Spacing

a. Special Characters: ( ) , space

There are four special characters; left parenthesis, right parenthesis, comma, and space. A symbol must be delimited on both ends by a special character. Additional spaces adjacent to a special character are ignored.

( ) Parentheses are used to enclose lists.

, Commas are used to separate arguments in a regular function call, in a TREET expression.

Space Spaces are used to delimit symbols when none of the other special characters is appropriate. They are also used to separate the function from its argument in infix notation.

b. Special Symbols

* The single asterisk is a special symbol which indicates that whatever follows is a comment. A comment is terminated by

another asterisk or by the end of the card.

** The double asterisk is a special symbol which terminates reading by the input routine in case of a parenthesis error.

/ A single slash or semi colon signifies the end of a statement, in a TREET expression.

// The double slash or colon is used to separate a label and its associated statement within the definition of a TREET coded function (i.e., in a DEF).

c. Spacing

The basic principle here is that a symbol must be preceded and followed by one of the special characters; i.e., by a left parenthesis, right parenthesis, comma, or space. Additional spaces adjacent to a special character are ignored.

(A) = (A) = (A) = (A)

Bearing the basic symbol principle in mind, spaces can be used freely, with the following exceptions:

(1) In a regular function call, the function name must be followed immediately by a left parenthesis.

TYPE (A) is illegal

TYPE(A) is legal

(2) In an infix function call, arguments must be separated from the function name by spaces.

4 + 6 is legal

4+6 will be interpreted as a symbol having 3 characters, not as a function call.

2. Compilation

By placing a C after the name of a TREET function a request is made for a compilation. A corresponding program in machine language will be generated. (See Appendix C)

3. TREET Function Types

The type of a function determines the way in which arguments are given to the function. Type R is standard; the others are seldom used except for system functions.

Type R — Type R functions expect a predefined number of arguments. For functions coded in the TREET language, that number is the number of symbols appearing in the argument list in the function definition. The arguments of a Type R function are evaluated before they are given to the function. Omission of type in a function definition is equivalent to specifying R.

Type U — Type U functions accept any number of arguments. As for Type R functions, the arguments are evaluated before they are given to the function. The several arguments supplied are collected in a list and given to the function as its first argument.

Type F — Type F functions expect a predefined number of arguments, as do Type R functions. They differ from Type R functions in that their arguments are not evaluated before they are given to the function. The functions themselves may, however, evaluate their own arguments.

Type FU — Type FU functions accept any number of unevaluated arguments.

4. Argument List and Variable List

A variable is a legal TREET symbol. Each variable must be preceded and terminated by a special character. Variables are bound by mentioning them in either the argument list or the variable list. Any previous value is automatically preserved and restored upon entry and exit from the function. If a variable is not bound at any level, it is considered a system variable.

At the entrance to a function, the variables in the argument list are set to their appropriate values. The variables in the variable list that are local to the routine are set to NIL, those that are global variables are not changed. These notions are probably best explained by example. Consider the following function (introduced solely to illustrate the different uses of variables):

```
DEF(   JOE   (A B)  (C D)
          C = A + 5/
L //      D = C + B + F /
          E = C - D /
          RET(E) /
          ) **
```

Variables used: A, B, C, D, E, F, L

Bound variables: A, B, C, D, L

Free variables: E, F

Arguments: A, B

Program variables: C, D

Location symbols: L

Before the function is entered, the values of A, B, C, D, and L are saved. A and B are set to the first and second arguments respectively. C and D are set to NIL. L is set to the address of the expression it stands for. E and F are not touched. As the program is executed, C, D, and E are set to new values. Upon exit from the function, the old values of A, B, C, D, and L are restored to their previous values. F was used but not changed. Note that E was set to a new value but was not restored. Its old value has been lost. Free variables are a way to communicate with the world outside (above) the present function. Bound variables are available only to the present function (in which they are bound) and functions called by the present function.

5. Label

A label is a TREET symbol followed by a double slash. A label may be attached to any statement for reference.

6. Statement

A statement is made of any or a number of functional expressions.

7. Comments

Comments may appear anywhere. The single asterisk is the special symbol which introduces the comment and either another asterisk or the end

of the card terminates the comment.

8. An Example Program

```
DEF(  COUNT   (X) (Y)     * COUNT NUMBER OF CELLS USED
      IF INTEGERP(X) THEN RET(1) /
      IF NUMBERP(X) THEN RET(2) /     * FLOATING POINT
      IF ATOM(X) THEN RET(0) /
      Y = 0 /
      WHILE(X, Y = Y + 1 + COUNT(CHOP(X)) ) /
      RET(Y) /
      ) **
```

This is a simple routine to count the number of cells in a list-structure. The function is called by using its name, (COUNT), with a list of arguments. Each symbol carries its own value(or is NIL), and its own pushdown list for that value. Recursion to any level is allowed in TREET. Note that an integer occupies one cell and that floating point takes two.

V. Predefined Functions

There are presently about 150 functions defined in the basic TREET package. Numerous others are available for special purposes, and, of course, the user may define as many as he wishes; system and user defined functions are not distinguished by the system. Predefined functions are categorized as list-processing, programming, arithmetic, control, or on-line.

A. List-processing functions

A fairly extensive set of functions is provided. Examples:

MEM1(list) - first member of a list

MEM2(list) - second member of a list

MEM12(list) - the first member of the second member of a list

REM2(list) - the remainder list after the first two elements have been removed.

CHOP(var) - value is the first member of the list that is the value of the variable and that variable is set to REM1 of the list.

LIST(a,b,c,...) - a new list containing the specified members.

B. Programming functions

Examples:

a = b - sets the first argument equal to the second

B(loc) - branch to the argument

PRINT(a, b, ..) - prints the values of the expressions separated by blanks on as many lines as necessary on the off-line printer.

TYPE(a, b, ..) - same as print except also types out on the on-line printer when operating in the on-line mode.

Other operations include the conditionals IF a THEN b and IF a THEN b ELSE c, where a, b, and c are arbitrary TREET expressions; branch; return (i.e. terminate this function and return the value of its argument); = (for setting variables); functions for grouping statements, loop control, etc.

C. Arithmetic functions

Arithmetic is performed by such functions as + (addition), X (multiplication), SIN (sine), LN (natural logarithm), INTVAL (integer value of), LT (less than predicate), GENRAND (generate a random number), etc. These functions take integers or floating point numbers as arguments. All arithmetic is done in floating point; results are converted back to integers if the result is integer. For example, the value of the TREET expression: SQROOT(5.0 - 2 X .5) would be (the integer) 2.

D. Control functions

Examples:

BINARY(list) - defines and stores compressed cards punched by defining functions.

TREET( )
READQ( ) } - controls the mode of the system

ONLINE( ) - initializes the system for on-line use.

DEF(any number of arguments) - defines TREET functions

E. On-line functions

Examples:

DT(tree) - display the specified tree (as a set of symbolic nodes connected by vectors) on the main part of the on-line display.

DLEFT(list) - display the specified list of atoms in a column in the left margin of the display.

DC(list) - display the specified list of cards on the main part of the display.

Note - Lightgun actions may be taken on these and other displays. These actions are decoded (providing for example, the address of the node pointed at) and cause a predetermined function to be executed.

A more complete list of predefined functions with their descriptions is found in Appendix B.

VI. TREET System Modes of Operation

The TREET system can be operated in either of two modes: TREET mode or READQ mode. Mode determines the way in which a command is handled. A command calls for the execution of some function; it can come either from cards or from the typewriter. Both the command format and the handling of the function's arguments are dependent on mode.

A. TREET Mode

In TREET mode argument evaluation depends on the function type. Arguments are evaluated for type R and U functions; they are not evaluated for type F and FU functions. The system accepts either regular or infix notation; multiple arguments of a regular function call must be separated by commas.

B. READQ Mode

In READQ mode the system expects every input (from cards or typewriter) to be a pair of elements. It expects the first element of the pair to be a symbol which has been defined as a function and the second element of the pair to be a list of arguments for the function. When the function is executed, the arguments are given, unevaluated, to the function. Thus in READQ mode, every function operates as if it were a type F function. Infix notation cannot be used in READQ mode. The expression TOT = 7 must be written =(TOT 7). Multiple arguments of a regular function call are not to be separated by commas.

Assuming A = 10, B = 14, and C = (X Y), in TREET mode the expression TYPE(A, B, C) will result in 10 14 (X Y); in contrast, in READQ mode the expression TYPE(A B C) will result in A B C.

C. System Operation and Mode Switching

The TREET System begins operating in the READQ mode when it starts its initialization - i.e., when it reads in the data deck. Function calls in the data deck must follow READQ mode conventions.

When the system goes on-line, it is initially in TREET mode.

The operator can switch the system mode online by typing

READQ() to put the system in READQ mode

or

TREET ( ) to put the system in TREET mode.

APPENDIX A

Augmented Backus Normal Form

The syntax of the TREET language is defined in an augmented Backus notation. To the regular Backus notation symbols, ::=, |, <, and >, the following have been added: [,], ^{0}, *, and ^{1*}. Their meanings are as follows:

::=	is defined as (assignment operator)
<name>	variables, where name is any string of characters
|	or (exclusive)
[	left bracket
]	right bracket
^{0}	zero or more occurrence of the preceding expression
*	zero or more occurrences of the preceding expression
^{1*}	one or more occurrences of the preceding expression

Any other symbol that occurs in a formula stands for itself. Thus the rule

<list-structure> ::= <atom> | (<list-structure>*)

means that a list-structure is either an atom or it is a left parenthesis followed by any number of list-structures followed by a right parenthesis.

APPENDIX B

TREET Function Descriptions

This Appendix contains descriptions of a set of basic TREET functions, those most likely to be required in using the TREET system. They will always be included in a TREET running system or deck.

The functions are classified as list-processing, programming, arithmetic, control, or on-line functions according to where they seem to fit best.

Most of the descriptions are followed by examples. The variable X, which appears in many of these examples, evaluates to the list

(A B (C D) E F)

The symbol → means "evaluates to"

The symbol ≡ means "is equivalent to"

1. List-processing functions

MEM1

R 1 argument

Value is the first member of its argument, which must be a list.

MEM1 (X) → A

MEM2
MEM3
MEM4

R 1 argument

Value is respectively the second, third, fourth member of its argument, which must be a list.

MEM2 (X) → B
MEM3 (X) → (C D)
MEM4 (X) → E

MEMmn

R 1 argument

Value is the m^{th} member of the n^{th} member of its argument, which must be a list. MEM11, MEM12, MEM21, and MEM31 are defined.

MEM21(Q(((A B) C))) → B

REM1

R 1 argument

Value is a list equal to its argument, which must be a list, but with its first member removed.

REM1(X) → (B (C D) E F)

REMN

R 2 arguments

The first argument must be a non-negative integer; the second must be a list. Value is a list equal to its second argument but with its first n (n=$argument_1$) members removed.

REMN(3, X) → (E F)

ADL add to list

F 2 arguments

The second argument must be a variable which evaluates to a list. ADL adds its first argument, evaluated, to that list and sets the variable equal to the augmented list. Value is NIL.

ADL('Z',X) → NIL

and

X = (Z A B (C D) E F)

The effect of ADL(a, b) is the same as b = CONS(a, b)

CHOP

F 1 argument

The single argument of CHOP must be a variable which evaluates to a list. The value of CHOP is MEM1 of that list. The effect of CHOP is to set the variable equal to REM1 of that list.

CHOP(X) → A

and

X = (B (C D) E F)

b = CHOP(a); is like:

b = MEM1(a);

a = REM1(a);

LIST

U Any number of arguments

Value is a list of its arguments.

LIST('A', 'B', 'C') → (A B C)

LIST('A', 'B', X) → (A B (A B(C D)E F))

EQUALS

R 2 arguments

EQUALS tests for the equality of the values of its two arguments. Value is true if they are equal. Numbers are EQUAL if they are within 10^{-13} of each other.

Q((C D)) EQUALS MEM3(X) → TRUE

REVERSE

R 1 argument

Value is a list having the same members as its argument (which must be a list); the members, however, are in reverse order.

REVERSE(X) → (F E (C D) B A)

CONS construct

R 2 arguments

The second argument of CONS must be a list.

The value of CONS is a list with the first argument as its MEM1, and the second argument as its REM1.

CONS('A', NIL) → ((A))

ALIST augment list

U Any number of arguments

Value is a list consisting of its last argument (which must be a list) augmented by its first n-1 arguments.

ALIST('A','B',X) → (A B A B (C D) E F)

ALIST(a, b) ≡ CONS(a, b)

MSL make single list

U

MSL forms a single list of atoms from its arguments.

MSL() → NIL

MSL(X)→ (A B C D E F)

APPEND

R 2 arguments

Both arguments of APPEND must be lists.

The value of APPEND is the (non-destructive) concatenation of both arguments.

X APPEND X →(A B (C D) E F A B (C D) E F)

MEMBER

R 2 arguments

Value is the sublist headed by the first equal member, if its first argument is equal to any member of its second argument, which must be a list. Value is NIL otherwise. (Member is usually used simply as a predicate.)

MEMBER('A',X) → (A B (C D) E F)

COPY

R 1 argument

Value is a copy of its argument (same structure but with all new list cells).

COPY(X) → (A B (C D) E F)

AND

R 2 arguments

Value is true if both arguments are <u>not</u> NIL; NIL otherwise.

LISTP(X) AND ATOM(MEM1(X)) → TRUE

OR

R 2 arguments

Value is true if either (or both) of its arguments is not NIL; NIL otherwise.

ATOM(a) OR LISTP(a) ≡ TRUE for any a

CARD

U CARD(ab...)

Card converts list structure to its print representation on cards.

Columns 1 to 64 of the card are used. Data on card 1 starts in column 1; on additional cards it starts at column 5. The standard right end carriage control is used.

Value of CARD is the generated card chain.

SOURCE

R SOURCE(c) or SOURCE(c func)

Value of SOURCE is NIL

The specified card chain <u>c</u> is scanned rather than the standard input (i.e., system card reader offline or the typewriter input buffer online).

When the source is exhausted, input will revert back to the standard source and the func, if any, will be applied to the empty argument list. An additional SOURCE will immediately terminate the current source; in particular, SOURCE() will immediately revert back to the standard input. Columns 1 to 72 of each card in the chain are used as they are when the card comes directly from the system card reader. Sequence numbers are checked in the same fashion.

SETPROP set property

R 3 arguments

SETPROP(a b c)

The first and second arguments must be symbols. SETPROP sets the property b of the symbol a to c. The value of SETPROP is the old property if any.

GET get property

R 2 arguments

Both arguments must be symbols.

Get retrieves the property b of the symbol a, if any. GET retrieves NIL if there is no such property.

GET(a b)

CARDP card predicate

R 1 argument

Value is true if its argument is a card; NIL otherwise

CARDP(X) → NIL

NULL
NOT

R 1 argument

Value is true if its argument is NIL; value is NIL otherwise.

NULL(X) → NIL

NULL(NIL) → TRUE

ATOM

R 1 argument

Value is true if its argument is an atom; NIL otherwise.

ATOM(5) → TRUE

ATOM(MEM1(X)) → TRUE

ATOM(NIL) → TRUE

LISTP List predicate

R 1 argument

Value is true if its argument is a list; NIL otherwise.

LISTP(X) → TRUE

LISTP(NIL) → TRUE

LISTP('A') → NIL

2. Programming Functions

=

F 2 arguments

= sets its first argument, which must be a variable to the value of its second argument. Value is the value of its second argument.

KEEP = X → (A B (C D) E F)

and KEEP is set to (A B (C D) E F)

B branch

F 1 argument

B causes a branch to its argument, which must be a location tag within the same function.

B(A1) → branch to location A1

IF conditional

FU 3 or 5 arguments

IF arg_1 THEN arg_3

or

IF arg_1 THEN arg_3 ELSE arg_5

The value of the IF function is the value of arg_3 if the value of arg_1 is not NIL. If the value of arg_1 is NIL, then the value of the IF expression is the value of arg_5 or, if there is no arg_5, NIL.

IF ATOM(X) THEN 5 ELSE 7 → 7

IFNOT
IFNULL

FU 3 or 5 arguments

{IFNOT arg_1 THEN arg_3
{IFNULL

or

{IFNOT arg_1 THEN arg_3 ELSE arg_5
{IFNULL

IFNOT and IFNULL are the same.

The value of the function is the value of arg_3 if the value of arg_1 is not true (i.e., if it is NIL). If the value of arg_1 is true then the value of the function is the value of arg_5; if there is no arg_5 then the value is NIL.

IFNULL Z THEN B(P1) ELSE B(P4)

WHILE loop control

F 2 arguments

Value is NIL. WHILE evaluates its second argument as long as its first argument evaluates to true (i.e., is not NIL).

INL = NIL;

WHILE(X, ADL(CHOP(X), INL)); yields

INL = (F E (C D) B A)

X = NIL

FOR loop control

FU 3, 4, or 5 arguments

Value is NIL. FOR evaluates its last argument over a range of a variable (its first argument).

If there are three arguments, the variable is stepped from 1 to the value of arg_2 in increments of +1.

If there are four arguments, the variable is stepped from arg_2 to arg_3 in increments of +1.

If there are five arguments, the variable is stepped from arg_2 to arg_3 in increments of arg_4, which must be a positive number.

```
FOR(A, 10, TYPE(SQROOT(A)))

FOR(A, 1, 10, TYPE(SQROOT(A)))

FOR(A, 1, 10, 1, TYPE(SQROOT(A)))
```

DO

U Any number of arguments

Value is NIL. The arguments of DO are TREET statements. DO is used to write a group of statements where only one is normally used. It is frequently used in WHILE and IF expressions.

```
IF A-C EQUALS 'F86' THEN

  DO( SPD = 480 / B(C1 /)
```

IFEQ if equal

FU Any odd number of arguments

IFEQ(arg_1, arg_2, arg_3.....arg_{n-1}, arg_n)

or

IFEQ(arg_1, arg_2, arg_3,....., ELSE, arg_n)

IFEQ tests its even numbered arguments, in order, for equality with its first argument. The value of the function is the value of the argument following the first argument which equals arg_1. If no equality is found, then the value of IFEQ is the value of the argument following ELSE; or if there is no ELSE, the value is NIL.

```
IFEQ(AC-TYPE,
   'F105', SPD = 880,
   'F100', SPD = 512,
   'F86',  SPD = 480,
  ELSE,    TYPE('ILLEGAL', 'AIRCRAFT', 'TYPE')
  )
```

Q quote

F 1 argument

Q means quote. The value of Q is its unevaluated argument, which may be any list structure. Q is used to cause operation on a symbol, not on its value; it is also used to create a list.

TYPE(X) results in the typeout of (A B (C D) E F)

TYPE(Q(X)) results in the typeout of X

Q((A B)) → (A B)

Single quote marks can be used instead of Q to quote a symbol, but not a list.

TYPE('X') ≡ TYPE(Q(X)) results in the typeout of X

PRINT

U Any number of arguments

Value is NIL. Effect is to print out its arguments on the system printer.

PRINT(X, 'IS', 'A', 'LIST') results in the printing of

(A B(C D) E F) IS A LIST

PRINT2

U Any number of arguments

PRINT2, PRINT3, & PRINT4 are the same as PRINT except that they space down one, two, or three lines, respectively, before printing.

TYPE

U Any number of arguments

Value is NIL. Effect is to type out its arguments on the typewriter. TYPE also prints its arguments on the system printer.

TYPE(X, 'IS', 'A', 'LIST') results in the typeout of

(A B (C D) E F) IS A LIST

TYPE2

U Any number of arguments

TYPE 2 is the same as TYPE except that it spaces down one line before typing out its arguments.

PUNCH

U Any number of arguments

PUNCH operates like PRINT but also punches out its arguments on the system punch. A + is punched in column one of each card so that the card will be interpreted by the facility. Data is punched in columns 2-72. Sequence numbers are punched in columns 78-80.

Value is NIL.

ERROR

U Any number of arguments

ERROR types out (and prints out) an error heading and its arguments. It then branches to the system ERR routine which terminates the current command, causes additional printout, and then continues with the next command.

ERROR(X) would cause the following typeout:

**** ERROR ****
(A B (C D) E F)

RET

R 1 argument

RET causes the function in which it appears to return (exit, finish) with the value of RET's argument as the value of the function.

RET(X)→ (A B (C D) E F) is the value of the function in which it is located.

3. Arithmetic functions

The arguments of arithmetic functions are assumed to be numbers. The number of decimal places printed out for the value of any function is determined by the setting of the variable PLACES.

+ Plus

R 2 arguments

Value is the sum of its two arguments.

1 + 2 → 3

X times

R 2 arguments

Value is the product of its two arguments.

10 X .5 → 5

- minus

R 2 arguments

Value is arg_1 minus arg_2

0 - 3 → -3

DIV divide

R 2 arguments

Value is arg_1/arg_2

1 DIV 2 → .5

ABVAL absolute value

R 1 argument

Value is the absolute value of its argument.

ABVAL($PI) → 3.14 ..

SQROOT square root

R 1 argument

Value is the square root of its argument, which must be a positive number.

SQROOT(2) → 1.414

ATAN arc-tangent

R 1 argument

Value is the arctan of its argument, expressed in degrees.

COS cosine

R 1 argument

Value is the cos of its argument, which is expressed in degrees.

COS(90) → 0

SIN sine

R 1 argument

Value is the sin of its argument, which is expressed in degrees.

SIN(90) → 1

INTDIV integer divide

R 2 arguments

Value is the integer part of arg_1/arg_2. $REM is set to the remainder

9 INTDIV 4 → 2 and

$REM = 1

EXP exponential

R 2 arguments

Value is the first argument raised to its second argument.

2 EXP 10 → 1024

10 EXP -2 → .01

INTVAL integer value

R 1 argument

Value is the least integer value of its argument.

INTVAL(4.74) → 4

INTVAL(20) → 20

LN logarithm

R 1 argument

Value is the natural log of its argument.

GENRAND generate random number

R No arguments

Value is a random number, between zero and one.

NUMBERP number predicate

R 1 argument

Value is true if its argument is a number; NIL otherwise.

NUMBERP(MEM1(X)) → NIL

INTEGERP integer predicate

R 1 argument

Value is true if its argument is a TREET integer; NIL otherwise.

INTEGERP(SQROOT(91)) → NIL

LEQ less than or equal to

R 2 arguments

Value is true if the first argument is less than or equal to the second argument; NIL otherwise.

7 LEQ 6 → NIL

GREQ greater than or equal to

R 2 arguments

Value is true if the first argument is greater than or equal to the second argument; NIL otherwise.

7 GREQ 6 → TRUE

GT greater than

R 2 arguments

Value is true if the first argument is greater than the second argument; NIL otherwise.

7 GT 8 → NIL

LT less than

R 2 arguments

Value is true if the first argument is less than the second argument; NIL otherwise.

6 LT 7 → TRUE

4. Control Functions

CLIST

R no arguments

CLIST turns on the off-line data-card-listing feature.

Value is NIL.

SLIST

R no arguments

SLIST turns off the card-listing feature and prints out the listing that has accumulated, if any.

Value is NIL.

BINARY

R 1 argument

The single argument of BINARY is a list. The compressed cards punched out by SP and DEF (and some other defining functions) are used as the "arguments" of BINARY. BINARY defines and stores the functions, variables, properties, etc., in accordance with these cards.

```
BINARY ((   .
            .
            .
      cards . output by SP
            .
            .
            .
      cards . output by DEF
            .
            .
            .
         )) .**
```

results in the definition and storage of the functions, variables, etc., according to the cards.

TREET

R no arguments

Puts the system into TREET mode.

Value is NIL.

READQ

R no arguments

Puts the system into READQ mode.

Value is NIL.

TWRITE

R 1 argument, must be an integer corresponding to a tape IOD.

TWRITE writes an image of core onto the specified type. This system (as it exists at this point) can be read in by the standard deck setup.

Value is NIL.

RESTART

R 3 arguments

RESTART(a b c)

Reallocates binary (a), pushdown (b), and card (c) space from skeleton core image tape. a, b, and c are integers which specify the number of words to be allocated.

RESTART must only be used on the empty (i.e., without data deck) system.

EVALL

R no arguments

EVALL prints out the values of all symbols that have values.

Value is NIL.

ONLINE

R no arguments

This initializes the system for online usage. No further data cards will be read. Each console is given a copy of TRS. Control is relinquished to background jobs.

Defining Functions

The defining functions are used to define functions and variable values. These functions should be used in READQ mode only. The examples show function calls as they would be written for READQ mode.

DEF

U Any number of arguments

The arguments of DEF, taken together, define a function in the TREET language. DEF translates this definition into Cambridge Polish and stores it as a list structure. DEF also punches out cards on which the function is defined in compressed form. These cards are subsequently used as arguments of the BINARY function.

SP Set and punch

U 3 or 4 arguments

SP sets pointers in the ATOMS table; it also punches cards acceptable to the BINARY function.

The first argument of SP can be S, =, or P.

If the first argument is S, then SP sets its second argument, which must be a variable, equal to its third argument.

SP(S X (A B (C D) E F))

X = (A B (C D) E F)

SP(S CEP 200)

CEP = 200

If the first argument is =, then SP sets its second argument, which must be a variable, equal to the value of its third argument (as opposed to the third argument itself).

SP(= DUP X)

DUP = (A B (C D) E F), assuming X had been set as shown above.

If the first argument is P, then SP sets the property, named as its third argument, of the variable named as its second argument equal to its fourth argument.

SP(P LIST EXPLAIN (LIST MAKES A LIST OF ITS ARGUMENTS))

sets the value of the property EXPLAIN of the symbol LIST equal to the following list:

(LIST MAKES A LIST OF ITS ARGUMENTS)

5. On-Line Functions

These functions are meaningful only when operating online.

DC display card

R 1 argument

DC(c)

The chain c (i.e., lists of cards) will be displayed in the center of the DD13, one card per line.

The value of DC is the sub-chain of cards that did not fit (normally NIL). The display will be made up from columns 1-64 of each card. Columns 65-80 are ignored.

The A6 characters slash (/), prime ('), dollar ($), and star(*), are displayed as quote ("). All other characters are displayed as themselves. A NIL in the argument list is equivalent to a blank card which is not displayed.

DT display tree

R 1 argument

DT(t)

Displays the tree t

The value of DT is NIL

DLEFT display left

R 1 argument

Displays a list of up to 25 words in a column in the left margin of the display.

Value is NIL.

DRIGHT display right

R 1 argument

Same as DLEFT, except the list of words is displayed in the right margin of the display.

DTOPB display top

R 1 argument

Displays the first five members of any list in a row across the top of the display.

This display is not subject to lightgun decoding.

DBOTB display bottom

R 1 argument

Displays the first five members of any list in a row across the bottom of the display.

This display is not subject to lightgun decoding.

PCHS print card highspeed

R 1 argument

PCHS(a)

The chain a (i.e., list of cards) will be printed on the highspeed printer.

The value of PCHS is the sub-chain of cards that did not fit (normally NIL).

PTH print tree highspeed

R 1 argument

PTH(a)

PTH prints on the highspeed printer a tree a from any list structure in tree format. Character values of the nodes are printed and connected orthogonally by horizontal and vertical lines.

Value is NIL.

APPENDIX C

The Compiler

I. Introduction

A complete compiling facility is available in TREET. A user can write a program in the TREET language and obtain a corresponding program in machine language. Previously, all TREET functions were translated to Cambridge Polish and needed the interpreter for execution. In order to obtain the speed advantage of compiled code it was necessary to code the function in TAP, the macro assembly language for the TREET system. The gap between Cambridge Polish (LISP) and TAP has now been closed by the compiler.

The principle advantage in compiled code is the speed advantage over the interpreter. (Basically each statement is decoded once at compile time rather than each time it is executed; thus for any statement that gets executed repetitively there is a significant speed advantage.) The TREET compiler can only be described as quick and dirty. That is, not much time was spent on it; it does a straightforward job (little or no optimization of any sort); the code it produces is dirty (a lot of steps are duplicated). Even so, it provides a significant advantage in speed. Preliminary tests indicate that compiled code runs about forty times as fast as interpreted code and that about the same amount of space is used. (Note, however, that TAP space rather than list space is used.)

II Use of the Compiler:

Compilation is requested by placing a C immediately after the name in the TREET function definition. The compile declaration if any must precede the type declaration if any. This will cause the ordinary Cambridge Polish compressed cards to be suppressed and TAP compressed cards will be punched.

Example:

```
DEF(  COUNT C  (X)  (Y)
        IF INTEGERP(X) THEN RET(1) /
        ..
        ..
        ) **
```

Compilation can also be requested directly for functions that already exist in Cambridge Polish. This is done by executing COMP(name) where name is the name of the function to be compiled. Example: (Assume COUNT is already in the system)

```
COMP(COUNT)
```

It is recommended that functions be compiled only after they are fairly well debugged. The interpreter provides much more error checking and more useful error diagnostics than the compiled version.

A printout of the TAP code produced by the compiler is produced if the free variable $PTL is other than NIL. The free variables PUN and DEFTDBG work as before.

An important requirement in use of the compiler is that all non type R functions called by the function being compiled must be declared. That is, their type must be known by the system at the time the compilation takes place. It is not necessary that the functions be defined in the way they will be used or that anything be known about their arguments. The type is specified at the time of definition; it may be necessary to put in a dummy definition of a function in order to specify its type for the compiler. For those users with a lot of non type R functions it may be easier to put in all the definitions first in the old way and call the compiler explicitly later on.

If TAP space is exhausted during a run, then a comment will be printed (OUT OF BIN PROGRAM SPACE) and further TAP functions cannot be entered. This does not affect the TAP cards that are punched. If this occurs during compilation of a TREET function, then that function will remain defined as an expression. (DEF always defines its function as an expression; if compilation is called for, it attempts to redefine it in TAP.)

The compressed card output from the compiler is about 2-1/2 times that of the Cambridge Polish format. $PFNUM is temporarily set to 200 by COMP.

III. Known Limitations of the Compiler:

1. The value of a DO expression is undefined (it is NIL in the interpreter). This means you cannot say A = DO (.. .. ..) and expect A to be set to NIL.

2. If an error occurs during the execution of a compiled function then the variables pushed down by that function will not be automatically restored. This may cause problems if a variable is used both as a system variable and as a bound variable; it can be circumvented by placing such variables on the variable list of a non-compiled function that surrounds the problem area.

3. A symbol cannot be used as both a variable and a location symbol within the definition of a function.

4. NIL will not be automatically substituted for arguments past the first in subroutine calls where they have been ommitted.

Examples: RET() ≡ RET(NIL) but EQUAL(A) ≠ EQUAL (A NIL)

5. A B or RET will not execute properly if it is embedded as an argument of a type F or FU function other than those provided for as special cases for the compiler. E.g., IF X THEN B(JOE) / is OK because IF is provided for, but CIF(B(JOE)) / where CIF is type F would not work because the CP expression (B JOE) would be given as an argument to CIF, which would not be able to execute the branch.

IV. Operation of the Compiler

It is not necessary to know how the compiler works in order to use it. However, the following commentary may be of interest, or of value, in using it to its best advantage.

The compiler consists of two principle functions (COMP and COMPE), a number of special cases for COMPE, and a number of auxiliary functions. COMP is sort of an executive; it finds the CP (Cambridge Polish) definition of the function specified to it, generates the top and bottom of the TAP list for bookkeeping, peels off an expression at a time and feeds it to COMPE, and finally calls DEFT (the TAP define function) to process

the definition. (DEFT punches the TAP cards.)

COMPE is the heart of the compiler; it takes a Cambridge Polish expression and from it generates the corresponding TAP code. COMPE is itself a recursive TAP function. It handles the general cases of atoms and functions itself and calls upon the DEFCC codes to handle special cases. (This split-up makes it possible to add new special cases without recoding COMPE and keeps COMPE to a reasonable size.) The translation that COMPE performs is specified in the next section. Two examples are provided in Figure 1.

V. COMPE Translations:

Terminology:

a - symbol; n - number; f - function; x, y, .. - arbitrary CP expressions

[x] - COMPE(x); GSi - generated symbol

a	(L a)
n	(L (Q n))
(f ..) where f is special - see below	
(f)	(Z $B1) (LINK f)

Type R functions:

(f x)	[x]	(ST $B1) (LINK f)
(f x a)	[x]	(ST $B1) (MUV a $B2) (LINK f)
(f x y ..)	[x] (PUSHA) [y] (PUSHA) .. .. (POP $B2) (POP $B1) (LINK f)	
	except replace innermost pair with (ST $Bn)	

TREET	Cambridge Polish:	TAP:	Comment:
A = B /	(= A B)	(L B)	load B
		(ST A)	store in A
IF A LT 5 THEN B(JOE) /	(IF (LT A 5) THEN (B JOE)	(L A)	translation of (LT A 5)
		(PUSHA)	
		(L (Q 5))	
		(ST $B2)	
		(POP $B1)	
		(LINK LT)	
		(BN L01)	branch on NIL to the generated symbol
		(B JOE)	
		L01	

Figure 1. Examples of COMPE Translation

Note: PUSHA puts the accumulator onto the top of the pushdown list. POP removes the top item from the pushdown list and puts it where specified by its argument.

TYPE U functions:

(f x y ..) [x] (PUSHA) [y] (PUSHA) ... (CADU n) (LINK f)

Type F functions:

(f x y ..) (MUV (Q x) $B1) (MUX (Q y) $B2) .. (LINK f)

Type FU functions:

(f x y ..) (MUV (Q (x y ..)) $B1) (LINK f)

Special Functions:

Expression	TAP Code
(= A x)	[x] (ST A)
(Q x)	(L (Q x))
(B A)	(B A)
(RET x)	[x] (CRET)
(MEM1 x)	[x] (CMEM1)
(CHOP A)	(CCHOP A)
(ADL x A)	[x] (ST $M1) (ADL $M1 A)
(WHILE x y)	GS1 [x] (BN GS2) [y] (B GS1) GS2
(IF x THEN y)	[x] (BN GS1) [y] GS1
(IF x THEN y ELSE z)	[x] (BN GS1) [y] (B GS2) GS1 [z] GS2
(IFNOT ...) same as (IF...)	except use BT instead of BN
(DO x y z ..)	[x] [y] [z] ..

```
(IFEQ A ... B2 E2 B1 E1 ELSE EO)
                    [A]   (PUSHA)
                    .
                    .
                    [B2] (LINK CIFEQ) (B GS3) [E2]   (B GS1) GS3
                    [B1] (LINK CIFEQ) (B GS2) [E1]   (B GS1) GS2
                    (POP) [EO]   GS1
```

(without else clause use (LI 0) instead of [EO].)

```
(FOR sym low high inc exp)
                  [low] (ST sym)
                   GS2 [exp][high] (ST $B2) (MUV sym $B1) (LINK GREQ) (BT GS1)
                   [inc]  (ST $B1)  (MUV sym $B2) (LINK +) (ST sym) (B GS2) GS1
```

Expression	Translated expression
(FOR sym high exp)	(FOR sym 1 high 1 exp)
(FOR sym low high exp)	(FOR sym low high 1 exp)
(GOTO A)	(B A)
(IFNULL ...)	(IFNOT ...)

APPENDIX D

TAP - The TREET Assembly Program

I. Introduction

TAP is a macro assembler for the TREET system. It is coded in SMAC, and was designed to code TREET problems in machine code. It accepts a symbolic specification for a set of functions and produces the corresponding binary programs directly in core. TAP does not punch cards and does not produce relocatable output. The results of the assembly always go into a reserved area of core. When the assembler has finished with a function definition, that function appears to the system as if it were coded in SMAC.

II. Function Format

The TAP assembler is a type R function with four arguments whose value is NIL; it has the following format:

DEFT(name-list type instruction-list)

arg 1: name-list is a list of names of functions to be defined. Each name in the name list must be defined as a location symbol. There may be any number of entrances (functions) defined with one DEFT command.

arg 2: type must be R, U, F, or FU. All names in the name list will be defined as functions of this type.

arg 3: instruction-list must be a list of location symbols and macro instructions as discussed below.

arg 4: normally NIL; if non NIL, the input to the second pass of the assembler and a dump of binary program space are printed.

Any symbol appearing in the instruction list will be defined as a location symbol whose value is the address of the next macro instruction. These symbols are defined for the current DEFT only. Location symbols may be used for both local storage and labels for machine instructions to be branched to (as in most assembly systems). They should not,however,be used for list structures which are not otherwise pointed to,as the garbage collector does not have access to them.

III. Operation of the Assembly

The TAP assembler consists of four parts: initialization, pass one, pass two, and finish.

A. Initialization:

1. Prints out the starting address in octal of the code produced followed by the list of names, argl.

2. Initializes several variables.

B. Pass One:

1. Defines location symbols

2. Converts the program list into a new list in which macros have been expanded and all location symbols have been removed. This conversion is accomplished by scanning through the list, defining and removing location symbols as they appear and replacing every operation that has a symbolic operation code by a copy of the list of operations into which the macro expands (each symbolic opcode must be defined as a macro).

The replacement process terminates when a numeric opcode is found; the next member in the list is then scanned. The result of pass one is a list of operation lists where the first member (if any) of each operation list is a TREET integer. If an undefined opcode is found, an error is printed, and the process continues with that operation ignored.

C. Pass Two:

Pass two accepts as input a list of operation lists of the following form:

(opcode address tag j-index length)

If the latter fields are not needed they may be left out.

The operation list specifies one half-word (32 bits) which is organized as follows and then stored in the appropriate half-word in core.

1. The opcode must be numeric and is stored into bits 8-31.
2. The address field if numeric is ored into bits 0-17;
   if it is a system or TREET symbol, the address of the symbol is stored into bits 0-17;
   if it is a local symbol, its address is stored into bits 0-18;
   if it is of the form (Q a) then the address of a word whose value field points to a is stored into bits 0-17.

3. The tag field must be numeric and ored into bits 28-31.

4. The j-index field must be numeric and is ored into bits 19-22.

5. The length field must be numeric and is stored into bits 3-8.

The following are TAP system symbols:

$LNK, $A1, $A2, $A3, $B1, $B2, $B3, $ANS, $FRE, $FND, $XWD, $DBG, $RET, $SAV

Any symbol used that is not a system symbol or is not defined as a location is considered to be a TREET symbol. A list of all TREET symbols used is printed out during assembly. TREET symbols used in TAP-coded functions are free variables (the same as free variables in TREET-coded functions).

D. Finish:

1. Prints out a list of all TREET symbols used.

2. Defines each name in arg1 as a function of the specified type as a SUBR starting at the address of the local TAP symbol.

3. Takes a dump if requested.

4. Returns NIL.

IV. TAP Macros

A. Format Description

The following is intended to provide a brief description of those macros that are likely to prove most useful in writing TAP functions. It is not intended that the user of TAP know the 7030 instructions which these expand into. There are about 125 macros defined in the system.

Macro instructions are written in the following form:

(opcode param1 param2...)

There may be zero to five parameters depending on the particular macro instruction. In any case where fewer parameters are coded than the macro provides for, zeroes will be substituted. If too many parameters are coded they will be ignored (no error comment).

A parameter may be:

1. a symbol
2. an integer. An integer parameter is taken as a (decimal) 7030 address or as a count or specification of an index register. This is <u>not</u> the same as a TREET integer. When used as a core address the number specifies a whole-word address (e.g., 9 is $R, 17 is $1.)
3. a floating point number. The integer part is taken as as above; the fractional part specifies a bit address.
4. a quoted expression of the form (Q a) where a is any list structure (TREET integers, floating point numbers, symbols, lists, list-coded trees). Note that 'abc' is equivalent (Q abc) where abc is an atom.

B. Basic TAP Macros

a, b variables

m, n variables which are set to lists

s location symbol (to be branched to)

i, j index registers

(ADL a m) Add to list adds a to the list m.

(AR4), (AR8), (AR16), (AR24), (AR32), (AR128)

Arrays of 4, 8, etc., members (count fields)

(ATOM a s) If a is atomic then branch to s.

(B s i) Branch to s($i).

(BN s) Branch on nil (accum 0).

(BT s) Branch on true (accum ≠ 0).

(CALL name) Link to the TREET-function name. Name can be any function except one defined by the current TAP definition (use LCALL for that case). CALL will set up a direct link or call the interpreter as appropriate.

(CONS a m b) Set b to the list m augmented by a.

(CNT m a) Put the count field of m into a.

(DAT m a) Put the daughter of m into a.

(EJECT a1 a2 a3 ..) Eject the page (1403) and print up to five things.

(ERROR a1 a2 a3 ..) Type **** ERROR ****

Type a1 a2 a3 ...

enter the system ERR routine.

(FREE) Put the address of a free cell into 14_{val}$.

(IF a s) If a is true (not NIL) then branch to s.

(EQ a b s) If a and b are the same symbol then branch to s.

(L a i) Load a($i). L(BU,18), a($i).

(LCALL name) Link to name defined within this DEFT.

(LI a) Load immediate a. LI(BU,18), a

(LISTP a s) If a is a list then branch to s. NIL is considered a list as well as an atom.

(LIST2 a b e) Create a list of two members a and b; put it into e.

(LIST3 a b c e) Create a list of three members a, b, and c; put it into e.

(MEM1 m a) Put the first member of m into a.

(MEM2 m a), (MEM3 m a), (MEM11 m a)
Put the second, third, first of the first member of m into a.

(MUV a b i) Move a into b($i). (Set b($i) to a)

(MUVQ a b) Move the address of a into b.

(NATOM a s) If a is not atomic then branch to s.

(NEG a s) Assuming a is a TREET integer then if a is negative then branch to s.

(NEMEM2 m s) If m does not have at least two members then branch to s.

(NEQ a b s) If a is not EQ (not the same symbol as) b then branch to s.

(NSYM a s) If a is not a symbol then branch to s.

(NULL a s) If a is NIL then branch to s.

(NUMP a s) If a is a number then branch to s.

(PAR m a) Put the parent of the node m into a.

(POS a s) Assuming a is a TREET integer then if a is positive then branch to s.

(PRINTn a1 a2 a3 ..) where n is blank, 2, 3, or 4
Print (on the 1403) a1 a2 a3,. after skipping 0, 1, 2, or 3 lines. There may be zero to five expressions printed.

(REM1 m a)	Put REM1 of m into a.
(REM2 m a)	Put REM2 of m into a.
(RPLACA m a)	Replace the MEM1 field of m with a.
(RPLACC m a) (RPCNT m a)	Replace the CNT field of m with a.
(RPLACD m a)	Replace the REM1 field of m with a.
(RPVAL m a)	Replace the VAL field of node m with a.
(RPDAT m a)	Replace the DAT field of node m with a.
(RPSIS m a)	Replace the SIS field of node m with a.
(RPPAR m a)	Replace the PAR field of node m with a.
(SAVEX loc)	Store the exit address into the branch instruction at loc.
(SIS m a)	Put the sister of m into a.
(ST a i)	Store into a($i). ST(BU,18),a($i)
(TYPEn a1 a2 a3 ..)	Does the same as (PRINTn ...) and also types out its arguments if online.
(VAL m a)	Put the value of node m into a.
(Z a)	Set a to NIL.

C. Creating New TAP Macros

TAP macros may be logically grouped into two categories: machine operations and machine independent (list-processing oriented) operations. TAP does not know the difference between these; indeed it is possible to code in TAP not using any macros at all. Basically, however, there is a set of macros that is designed to be used directly and another set in terms of which other macros are defined (to be used directly only when necessary for some special purpose).

Most of the former are actually defined in terms of the latter, but this is not necessary and in some cases is rather difficult (without creating an ad hoc macro). The former are called TREET macros and the latter STRETCH macros. In general STRETCH macros are similar to the corresponding STRETCH instruction. For example, (LX 2 A) corresponds to LX,2,A.

These macros must be defined in a format acceptable to pass two of DEFT2. The opcode is specified numerically and the remaining fields must correspond to the fields as defined under DEFT2. A numeric opcode in a macro definition is taken as an octal number (it is read in as a normal decimal integer but its binary representation is changed by the function MACRO). This is not the case if a numeric opcode is used directly in a TAP-coded function.

A macro may have zero to five parameters. If a parameter is called for but is not filled in when used, it is taken as 0.

A macro may expand into any number of additional macros and/or operations.

There is at present no facility for: (1) relative addressing; (2) specifying the current location; (3) location symbols within macros (although they can be used once correctly, if the same macro were called twice in the same definition there would be a doubly defined (ambiguous) symbol).

To define macros use MACROL ((.. .. .. ..)).

APPENDIX E

TREET Floating Point Numbers

I. Introduction

A TREET floating point number is a STRETCH floating point number coupled with an auxiliary word that contains information necessary for list processing. This auxiliary word is in index format. Its value field contains the address of the number and its count field contains a 4, identifying it as a TREET floating point number. TREET floating point numbers are made up for any numbers containing a decimal point input as TREET atoms, for all integers greater than 2^{23} -1, and for all results of the arithmetic functions that either are not integral or exceed 2^{23} -1. While practical limits have been imposed on the input and output of TREET numbers, the full precision (approximately 14 decimal digits) of the machine is maintained during calculations.

II. Input

Upon discovery of a decimal point in an atom (say a number typed in by a user) the system constructs a floating point number for that atom. It does this by first taking the integer part of the number, converting it from decimal to binary, and storing it as the fraction field of a floating point number that is given an exponent of 48 and normalized. The fraction part of the atom is then converted from decimal to binary and stored as the fraction field of another floating point number with an exponent of 48 and normalized. This second floating point number, since it is really a fraction but stored as a whole number, is divided by 10^n, where n is the number of digits to the right of the

decimal point, to restore it to its true value. These two STRETCH floating point numbers then represent the integral and fractional parts of the atom. They are added, using floating point addition, and their sum is stored as the TREET floating point number. If the atom has a minus sign, the fraction sign of the floating point number is set negative. For Ø, a floating point word of all zeros is created.

Input is limited to 10 characters but full precision can be obtained by scaling. For example,

```
TMT = .00001 X .00001 / * TEN TO THE MINUS TEN
PI = 3.1415926 + 535.8979 X TMT /
```

is a method of entering PI to 13 decimal places and accurate to 14 digits.

Note that the setting of the variable PLACES has no effect on input.

III. Output

For output, a floating point number is first multipled by 10^X, where X equals PLACES, to convert it to an integer in the accumulator. PLACES is a TREET variable, initially set to 4, that can be changed by the user. Its setting determines the number of places to the right of the decimal point that will be printed. The number printed is rounded to the number of places by the addition of $5/10^X$ where X = PLACES + 1. An add double of a word with an exponent of 48 and a fraction of 0 moves the rounded floating point number into the high-order fraction field of the accumulator. This integer, which really also contains the fraction, is converted from binary to decimal and a decimal point is inserted X places from the right where X equals the current value of PLACES.

If, for example, the two statements illustrating the input of PI were part of the definition of a function, and PLACES was set to 4 when the function was processed, the function would have been correctly defined but the numbers punched on the compressed output deck would have had only 4 decimal places. They would have been .0000, .0000, 3.1416, and 535.8979 respectively. These numbers, especially the zeroes, are unsuited for re-entry into the machine. To have been punched out fully, PLACES should have been set to 7 or more when the function was processed. With PLACES set to 7 the last number would be punched as 535.8979000, which is 11 characters. On re-entry it will cause an "illegal atom" comment but will be interpreted correctly.

Full precision is provided automatically on output for numbers less than 2^{48} and greater than 1 if PLACES is set properly. The effective value of PLACES is reduced if necessary so that output will have, at the most, 15 significant decimal digits. For example: with PLACES set to 9, the sum of 1234567890 and .123456789 is 1234567890.12346 .

To reiterate, output is rounded based on PLACES. For example 1.8 put in with PLACES set to 0 (meaning, print no places to the right of the decimal point) prints out as 2. because of rounding. Rounding affects only output; the internal floating point number is not changed.

IV. Integers Used by Arithmetic Functions

All the TREET arithmetic functions use floating point arithmetic, making necessary the conversion of all integer arguments for these functions to floating point. This is done by loading the integer value from the index word and storing it right-adjusted into the high-order

fraction field of the accumulator and giving it an exponent of +48. The integer's sign goes into the sign byte register. The contents of the accumulator is then stored as a normalized floating point number. Integer zero is converted to a floating point word with a zero fraction and an exponent of -1023.

V. Results of Arithmetic Functions

The floating point result of any arithmetic function is converted into a TREET integer if it is an integer and not greater than $2^{23}-1$. This is done by double adding to the result a floating point word with a zero fraction and an exponent of 48 and testing the low-order fraction field for 0. If the test fails, the number is not an integer and it remains floating point. If the test succeeds, the number is an integer whose value is the 24 low-order bits of the high-order fraction. This value is stored into a TREET integer skeleton word whose sign is set from the sign byte register.

Results that are mixed numbers or fractions and results that exceed $2^{23}-1$ remain floating point.

VI. Miscellaneous Comments

TREET floating point numbers have one idiosyncrasy. The reclaimer function examines bit 0 of all words in list structure to see if it has been set to 1, indicating that this word is no longer in use. For floating point words this conflicts with the usage of bit 0 as the exponent flag bit. What this means to the user is that in the extremely rare cases that he computes a result falling outside the range of 2^{+1023} and 2^{-1023} the result, because the exponent flag bit is set to 1, appears to TREET to be marked for reclaiming and is not saved.

Any integer greater than $2^{23}-1$ input without a decimal point is

converted to a floating point number because a TREET integer, being a value field whose first bit is used by the reclaimer function, is limited to 23 bits.

Some decimal fractions cannot be represented precisely in the binary system. This causes some arithmetic results, which are integers in the decimal system, to be mixed numbers in the binary system; that is, they only approach being integral. Results that are exact floating point integers are printed with the decimal point but without the fractional zeroes regardless of the value of PLACES.

VI. Limits

The table below shows the limits for input and output of TREET integers and floating point numbers.

LIMITS

INPUT	OUTPUT
10 BCD CHARACTERS	17 BCD CHARACTERS
INTEGERS ± 0 to ± 8388607	INTEGERS ± 0 to ± 8388607
FLOATING POINT .000 000 001 to 9 999 999 999 (one less place with sign.)	FLOATING POINT ± .00000000000001 to ± 281 474 976 710 655.

APPENDIX F

Garbage Collector and List Storage

The TREET system contains an automatic garbage collector for reclaiming list cells and cards that are no longer needed. Originally all unused space between the upper limit of programs and the lower limit of common is made into the list storage area. All cells are linked together and put on the free storage list(FRL). As cells are used they are removed from the free storage list. When the free storage list is exhausted a reclaimer occurs. The reclaimer first marks all cells that can be reached and then does a linear sweep of the list storage area, collecting all cells that have not been marked. The marking bit (bit 0) is erased at this time. Cards are linked, marked, and recovered in the same fashion.

The reclaimer may also be called explicitly by RECLAIM(). The reclaimer prints out the number of cells and the number of cards collected. The initialization program prints out the original size of list storage. Normally all list cells, all integers, and both words associated with floating point numbers are stored in the list storage area. The individual atom pushdown lists are also stored in this area.

THE MITRE CORPORATION